

DEVELOPING A SCHEDULING SYSTEM IN ADA - PROBLEMS AND LESSONS LEARNED

Stuart Weinstein, Carol Whitney, David Zoch
Loral AeroSys
7375 Executive Place, Suite 101
Seabrook, MD 20706
(301) 805-0456

G. Michael Tong
Data Systems Technology Division, Mail Code 522
Goddard Space Flight Center
Greenbelt, MD 20771
(301) 286-3176

ABSTRACT

The Request-Oriented Scheduling Engine (ROSE) is a scheduling system written in Ada. Initially, ROSE was written in LISP, but it was redesigned in Ada to conform with NASA's movement toward Ada. Although Ada supports many object-oriented design features, problems are encountered in implementing an object-oriented design. Furthermore, documenting an object-oriented design is not straightforward. Our approaches to these problems are described. Also, our lessons learned in the use of Ada are presented.

INTRODUCTION

The Request-Oriented Scheduling Engine (ROSE) is a product developed by the Data Systems Technology Division of the Goddard Space Flight Center, which promotes technology efforts to investigate state-of-the-art software methods and new approaches. We selected Ada as an implementation language because the use of Ada could become mandatory for many NASA data systems.

Several topics are covered. First, an overview of ROSE is presented. Second, ROSE's object-oriented design is shown. Third, a documentation utility and reusable data management routines are described. Fourth, we show that an artificial intelligence algorithm can be implemented in Ada. Finally, a lessons learned section describes our experiences on the use of Ada.

ROSE OVERVIEW

ROSE supports spacecraft operations scheduling. A spacecraft contains many flight instruments, on-board computers, and engineering subsystems that provide attitude control, power, thermal control, communications, and data management. The scheduling software must keep track of items that affect spacecraft operations. Additionally, timing relationships are maintained between spacecraft activities and between an activity and an orbital event, such as spacecraft sunrise.

ROSE has a general framework that provides a blueprint for implementation and a model for accomplishing scheduling actions. In particular, ROSE supplies a model for performing the following functions: specifying user resource requests, managing numerous schedules, incorporating different scheduling algorithms, manipulating objects on the graphics screen, managing schedule changes, validating schedules, and implementing advanced user interfaces that keep up with technology upgrades. ROSE provides specific tools to instantiate the general model. Most of the application-unique knowledge is represented as data in the scheduling language called Flexible Envelope Request Notation (FERN). The scheduling language allows users to tailor requests to their applications by creating user-defined time segments.

Figure 1 shows ROSE's Timeline Manager Screen which can display several schedules for visual comparison. On the timeline plot, rectangles represent scheduled activities, and histograms show resource usage. Timeline functions include edit, cut, paste, dragging activities, repositioning timelines, scroll, and zoom. The operator may slide the activity rectangles around to perform rescheduling by direct manipulation.

Additional displays, located below the timeline plot, show resource conflicts, constraint violations, and schedule status. The operator may change these displays by making selections

Figure 1. Timeline Manager Screen
from a menu. As the operator slides an activity on the timeline, the bottom displays are updated in real-time to show scheduling status and conflicts.

The first work on ROSE began in May 1987. An initial scheduling prototype was written in LISP/Zetalisp and executed on the Symbolics 3600-class computer. In June 1989, development of the Ada version of ROSE started. Version 1.2 was delivered in December 1991.

The Ada version of ROSE executes on either UNIX or VMS, thus achieving a certain level of portability. If one defines lines of code by counting semicolon statements, ROSE has 26,000 lines of code.

ROSE'S OBJECT-ORIENTED DESIGN

Ada packages support the ROSE object-oriented design. A package encapsulates an object's data and provides a set of predefined operations to manipulate and access that data. However, Ada's scoping and typing rules cause some problems in encapsulating objects. Ada provides incomplete type declarations and private types. The *Ada Reference Manual* states: "If the incomplete type declaration occurs immediately within either a declarative part or the visible part

of a package specification, then the full type declaration must occur later and immediately within this declarative part or visible part." Thus, mutually dependent and recursive access types cannot be placed in separate packages but must be gathered in one package. This problem prevented us from implementing an object-oriented design that has only one object per package. For example, instead of placing three objects in three separate packages, Ada may require that the three objects be gathered into one large package. Furthermore, in one case, an UNCHECKED_CONVERSION function was needed to get around the problem.

Drawing a diagram of an object-oriented design is difficult. Documenting a typical functional design is easier because there are standard techniques. For example, structure charts show the hierarchical relationships among modules. However, a structure chart does not work well for an object-oriented design because relationships between objects are not hierarchical.

We designed two diagrams to show relationships between objects. The first diagram, Figure 2, depicts subordinate relationships. An arrow from Object A to Object B indicates that Object B is subordinate to Object A. "Subordinate" implies that a worker/supervisor relationship exists between B and A. Arrows at both ends signify that either object may be subordinate depending upon the specific procedure or function. The diagram organizes objects into a (loose) hierarchy; however, the arrows go in many directions rather than from top to bottom.

The second diagram, Figure 3, shows the external calls made between objects. An arrow from Object A to Object B indicates that A makes external procedure or function calls to B. External calls differ from a subordinate relationship. A subordinate relationship implies a tighter coupling for both data and control, and a conceptual linkage.

ADA CALLING-TREE TOOL (ACT)

Because we could not find suitable Ada documentation and reverse-engineering tools, we developed a documentation utility named the Ada Calling-tree Tool (ACT). ACT produces calling-tree listings for Ada programs. It lists all external calls to and from a given subprogram. An Ada subprogram is a procedure or function. ACT is written in Ada and executes in Digital Equipment Corporation's (DEC's) VAX/VMS environment.

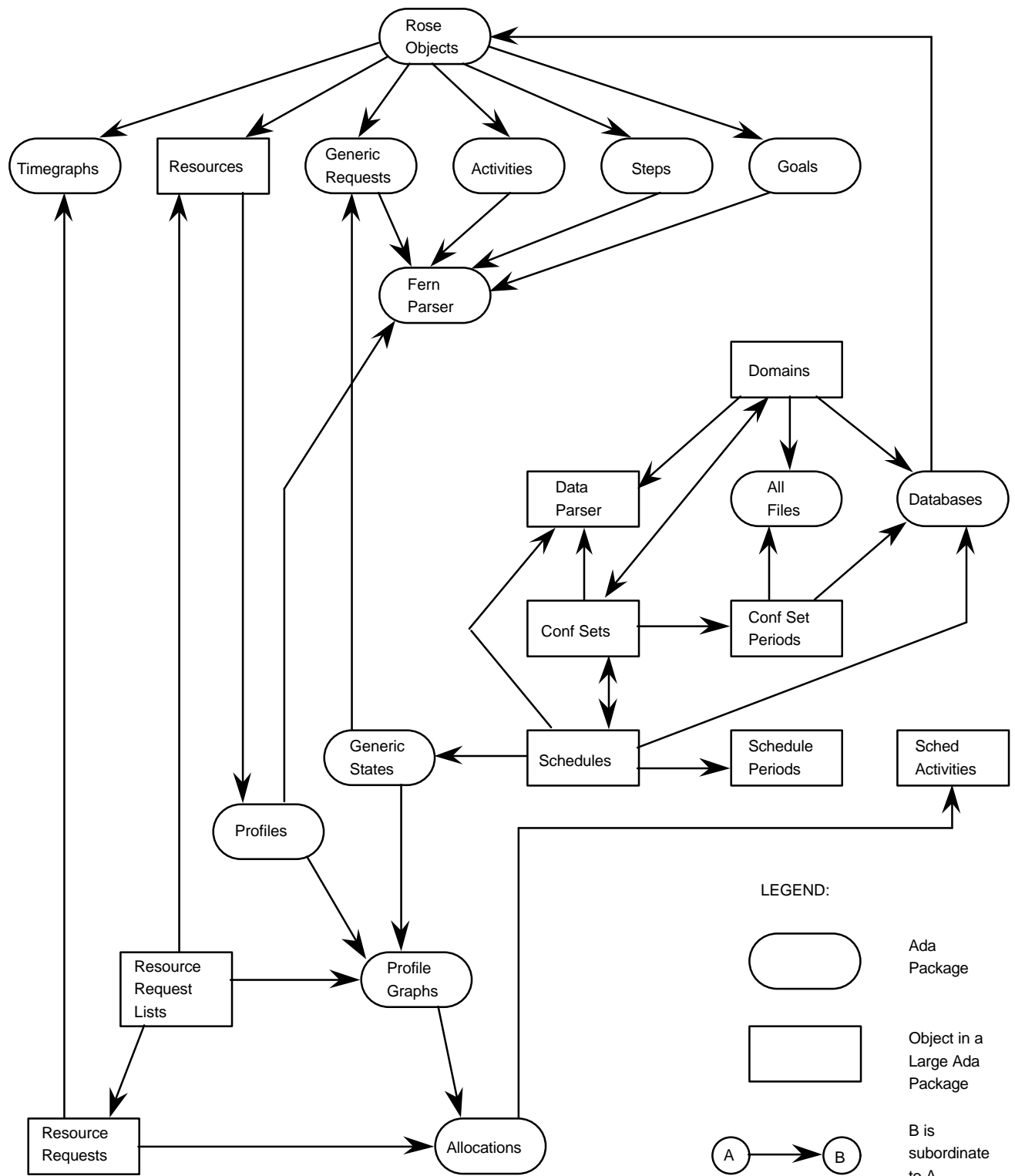


Figure 2. Objects with Subordinate Relationships

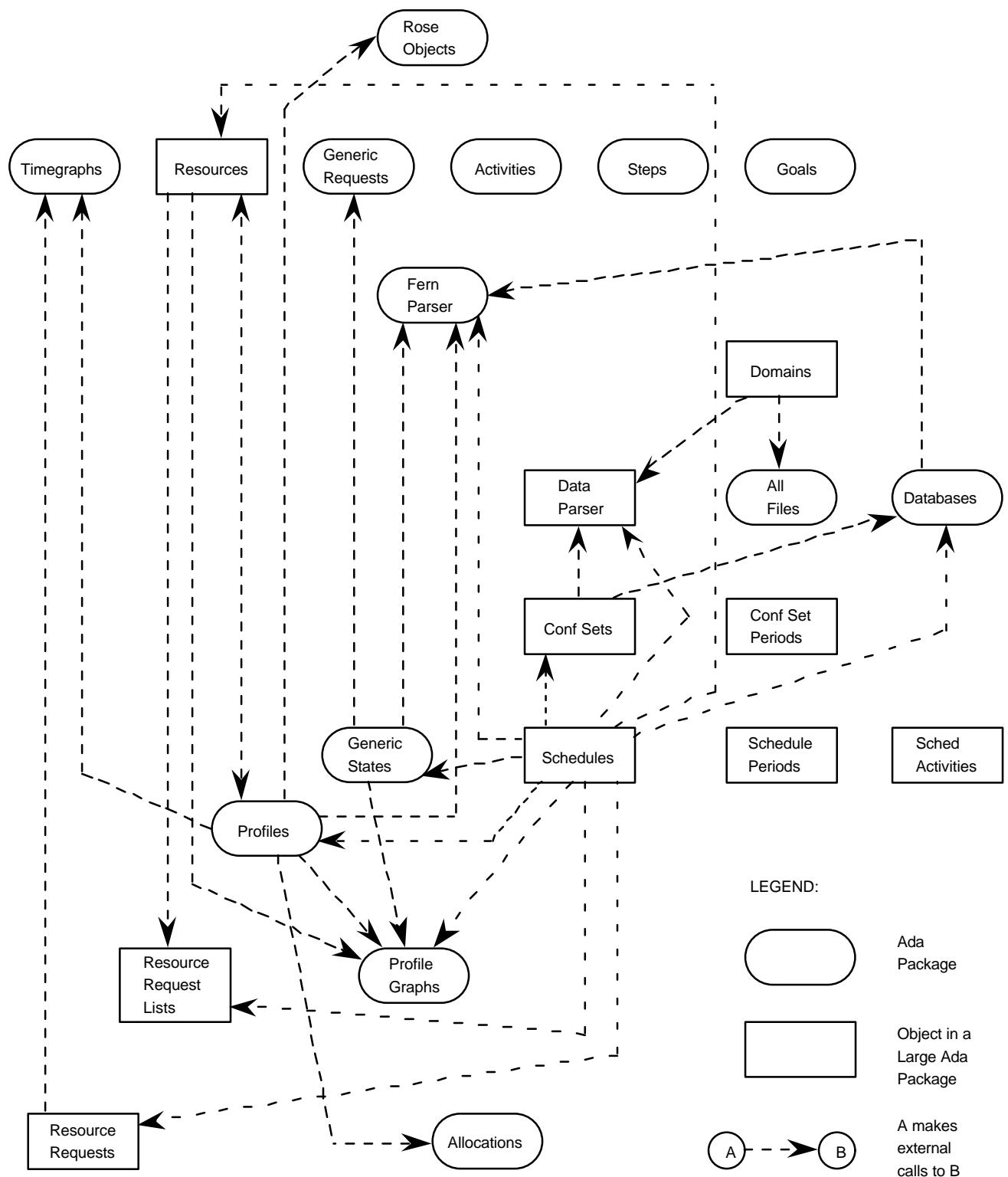


Figure 3. Objects with External Calls

Calling-tree listings are especially useful for analyzing an object-oriented design. With an object-oriented design, a programmer often sees just a list of functions and procedures rather than a sequential flow of executable statements. Calling trees show how the procedures and functions are invoked.

ACT performs functions similar to DEC's Source Code Analyzer (SCA). Initially, we tried to use SCA to generate calling-tree listings. However, SCA strips off the Ada package name and keeps the procedure or function name only. SCA cannot produce calling trees where each procedure or function is prefaced with its package name. For example, SCA represents an external call to "DATABASE.Create" only as "Create". Without the package name, it is difficult to locate the procedure or function in a large Ada program.

If the procedure or function is polymorphic, then the arguments of the procedure or function need to be examined for resolution. Polymorphism is a capability that allows many procedures or functions to have the same name. In Ada, the common term for polymorphism is "overloaded." An overloaded procedure or function causes SCA to produce ambiguous results. For example, the procedure "Create" may be defined in many packages.

DEC's technical representatives are aware of SCA's deficiencies with respect to Ada software. They indicated that future versions of SCA will have better Ada support.

REUSABLE DATA MANAGEMENT ROUTINES

We produced a library that contains general-purpose, data-manipulation software designed for reuse. Generic packages include variable linked list routines, dynamic length string functions, routines to access data stored by hash tables, and ring data structure functions. Other packages include time conversion routines and sort programs.

IMPLEMENTING AN ARTIFICIAL INTELLIGENCE ALGORITHM IN ADA

Artificial Intelligence (AI) is an emerging technology that includes rule-based systems, image processing techniques, and heuristic search algorithms. In general, AI algorithms try to solve problems where the human produces better results than traditional computer algorithms. AI algorithms may or may not simulate a human problem-solving approach.

Certain functions, such as list processing, must execute efficiently for AI algorithms to be practical. Ada does not have an AI heritage; however, it can support AI algorithms. AI algorithms can be implemented in any general-purpose programming language, but excessive execution times may become a problem.

ROSE has a rescheduling algorithm that uses a classical AI technique called "Best-First Search." The algorithm is written in Ada, and it employs heuristics and evaluation functions to

control and limit the search for rescheduling solutions. The remainder of this section explains the algorithm for readers who are interested in AI techniques.

Scheduling involves placing activities on a schedule. If an activity does not currently fit on the existing schedule, the rescheduling algorithm tries to adjust the schedule so that the activity can fit in the schedule.

The rescheduling algorithm uses a heuristic search. The search method provides a strategy for finding a solution where the problem space is extremely large. In traditional computing methods, algorithms are designed to provide a solution in a direct manner. However, traditional methods become impractical when a vast number of combinations must be evaluated. The heuristic search method does not evaluate all of the combinations. Instead, it chooses a path and then estimates the cost to get to a solution. The estimates are fed back to the algorithm. Because estimates are used, the algorithm may fail to find a solution.

The heuristic search method employs a directed graph that uses the following terms: state, node, edge, and goal state. We define the terms and then show an example. A state is a particular configuration of the problem elements. A node represents a state in the problem space. An edge (shown as an arrow in Figure 4) represents an action that transforms one state into another state. A goal state is a solution to the problem.

The root node of the graph represents the initial state. A solution is found when a path connects the initial state to a goal state. Each node contains an estimate of the cost to get to a goal node. A heuristic generates the estimate so the quality of the heuristic determines the effectiveness of the algorithm.

Best-First Search is a particular method of heuristic search. Figure 5 shows the pseudo-code for the Best-First Search in its simplest form.

Now, we show an example where ROSE employs the Best-First Search to perform rescheduling. A node represents the state of the schedule. The initial state has two problem elements: the schedule and an activity not on the schedule (the unscheduled activity). A goal state also has two problem elements: the adjusted schedule and the unscheduled activity successfully placed on the schedule. An edge represents the addition or deletion of activities on the schedule.

A resource deficit occurs when an activity's resource needs exceed the amount available. The rescheduling algorithm examines the timeline and selects a specific time when the unscheduled activity's resource deficit is small. Activities that are already on the schedule are removed until the unscheduled activity can fit on the timeline without any resource deficits. Then, the activities—those that were removed—are placed back onto the schedule, using the same evaluation functions.

Figure 4 shows a specific example of the Best-First Search method. Figure 6 shows the pseudo-code for the rescheduling algorithm. The search graph has three different types of nodes. The first type, SCHEDULE node, represents a state that has a single unscheduled activity that needs to be placed on the timeline. The first SCHEDULE node represents the initial state. The

successors of SCHEDULE nodes are AT nodes. They represent specific times where we will try to place the unscheduled activity onto the timeline. The successors of AT nodes are AND nodes. Each AND node represents a group of activities that were removed from the schedule so that the unscheduled activity could be placed at the time specified by the parent AT node. The hierarchy repeats where the children of AND nodes are SCHEDULE nodes. The SCHEDULE nodes represent the activities that were removed by the parent AND node. We will try to fit each of those activities back onto the schedule.

In Figure 4, an edge from a SCHEDULE node to an AT node represents adding an activity to the schedule. Edges from an AT node to an AND node represent deleting activities from the schedule. Edges from an AND node to a SCHEDULE node represent choosing an unscheduled activity.

We augmented the Best-First Search method to process AND relationships in addition to the mandatory OR relationships. The Best-First Search method employs a logical OR relationship

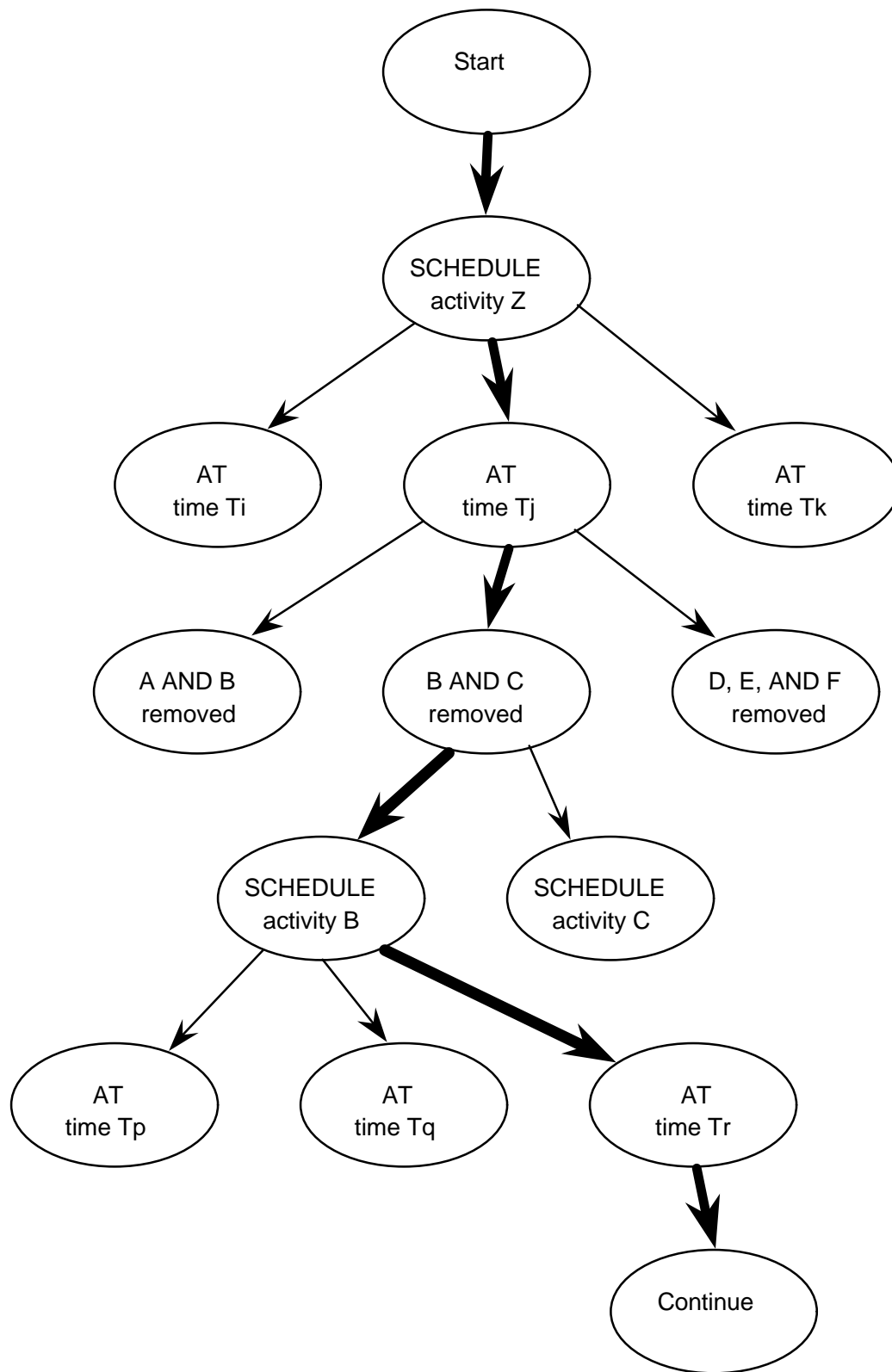


Figure 4. Rescheduling Search Graph

```

Create a search graph G, initially containing only the start
    node S;
Put S on a list called OPEN;
while OPEN is not empty loop
    Remove node N with the lowest cost from list OPEN;
    if N is a goal node then
        exit with SUCCESS; -- the solution is the path from S to N.
    end if;
    Generate the successors of N in G;
    Add these successors to OPEN;
end loop;
exit with FAILURE;          -- no solution was found.

```

Figure 5. Best-First Search Pseudo-Code

```

Create a search graph G, initially containing only the start
    node S, which is a SCHEDULE node for the
    unscheduled activity;
Put S on a list called OPEN;
while OPEN is not empty loop
    Remove node N with the lowest cost from list OPEN;
    if N is a goal node then
        exit with SUCCESS;
    elsif N is a subgoal node then
        Remove from OPEN all siblings of N, and all
        descendants of those siblings;
        Add the next sibling of N's parent to OPEN;
    else
        if N is a SCHEDULE node then
            Generate successors that are AT nodes;
            Add all successors to OPEN;
        elsif N is an AT node then
            Generate successors that are AND nodes;
            Add all successors to OPEN;
        elsif N is an AND node then
            Generate successors that are SCHEDULE NODES;
            Add first successor to OPEN;
        end if;
    end if;
end loop;
exit with FAILURE;

```

Figure 6. Rescheduling Algorithm Pseudo-Code

among the edges emanating from a node. The logical OR relationship requires only one solution path. On the other hand, a logical AND relationship requires solution paths for all of its children.

An OPEN list is employed to implement the AND relationship. The algorithm allows only one child of an AND node to be on the OPEN list at a time. But the cost estimate of that node is adjusted to include the cost estimates of all the other children of the AND node. If a solution for the child node is found, then the next child of the AND node is added to the OPEN list.

A heuristic generates the cost estimate which measures the difficulty of the task specified by the node. The SCHEDULE, AT, and AND nodes use different heuristics and input data to calculate cost estimates. For the SCHEDULE node, the input data are temporal constraint restrictions and resource availability. The input data for the AT node are the unscheduled activity's resource deficit at the specified time and the temporal constraint restrictions of the scheduled activities at that time. The cost estimate of an AND node is the sum of its SCHEDULE nodes' cost estimates.

A solution to the search requires an AT node with a cost estimate of zero. A zero cost means that the unscheduled activity can be placed on the timeline at the specified time without moving any other activities. If the AT node is a goal node, then the algorithm successfully found a solution. Otherwise, the AT node with zero cost is called a subgoal node.

An AT node is a goal node if all of the AND nodes on the path between it and the initial state are satisfied. This condition is a goal state that has two problem elements: the adjusted schedule and the unscheduled activity successfully placed on the schedule. In the goal state, all activities that were removed are successfully reinstated on the schedule. This condition occurs when an AT node completes the solution for the right-most (last) schedule node of one top-most AND node. Note that one child AT node can satisfy its parent SCHEDULE node, but all children SCHEDULE nodes must be successful to satisfy their parent AND node.

Finally, we need to use control parameters to abort the search that may never end. Control parameters include a maximum number of iterations of the "while" loop and the number of successors generated at each iteration. If no solution is found after the maximum number of iterations, the state of the search is saved so that the search can be resumed later.

LESSONS LEARNED

Our lessons learned list is divided into three parts: effective Ada features, problems encountered, and LISP/Ada comparisons.

The effective Ada features include standardization, generic units, package specifications, and dynamic memory management.

- Ada's standardization supports portability and reuse. Identical versions of ROSE execute on VAX computers under VMS and on Sun, HP, and DEC workstations under

Unix-like operating systems. In developing ROSE, we obtained software components from public software repositories and Loral's reuse library.

- Ada's generic unit is a robust construct. Generic units provide an efficient implementation of ROSE's object-oriented features. In particular, many of ROSE's graphical user interface objects use generic units. Because generic units are instantiated at compile time, the compiler can make optimization decisions.
- Ada's separation of a program package's specification from its body helps to enforce good software engineering practices. The encapsulation of essential interface information that is independent of implementation detail leads to better abstractions and hiding of low-level design decisions.
- Ada supports explicit control over dynamic memory management. ROSE's Ada run-time performance is more predictable than its LISP-based predecessor. Also, when an object is de-allocated in Ada, the pointer is automatically set to null. This feature greatly enhances debugging of pointers. As a side note, Ada does not guarantee that unreferenced, dynamically allocated memory (garbage) will be collected for re-allocation.

Problems encountered in the use of Ada include Ada's lack of certain object-oriented programming features, weak event-driven support, and mediocre debugging tools.

- Ada (specifically, Ada83) does not offer the complete object-oriented programming features found in C++, LISP Flavors, and other programming languages. The missing features include explicit inheritance, dynamic binding, and the ability to pass procedures as parameters. Furthermore, in our implementation of objects as packages, objects referenced each other. These parent/child relationships could not be implemented in Ada using the same object-oriented design as in the LISP prototype.
- Ada's step-wise refinement approach has weak event-driven support. The step-wise refinement design philosophy is inherent in Ada's constructs for nested packages (subpackages) and nested programs (subprograms). However, event-driven applications, such as ROSE's mouse-driven X Windows user interface, do not fit well with this philosophy. The step-wise refinement design does not produce a good map of the data structures for numerous pull-down menus and selectable objects that are displayed on the user interface.
- Ada needs better debugging facilities. Inspecting program execution was more difficult compared to the LISP environment. Many Ada debuggers lack symbolic reference to private types and direct support for objects that use generic instantiation, overloading, renaming, or nested scopes. Debugging by print statements is cumbersome since one needs to instantiate print routines for each user-defined data type. Private types and unconstrained arrays require type conversion before printing. Because conditional compilation features are not supported, adding print statements for debugging requires modifying code, not only for adding them but also for removing them.

Our comparisons of LISP versus Ada cover list processing, dynamic data structures, variant records, incremental compilations, and salient LISP features.

- List processing is straightforward in Ada with two caveats. First, memory management is explicit. Shared structures may have data access problems, but reference counters can be used. Second, pointers are typed in Ada. Dynamic variant records may be used; however, problems may occur (see below).
- LISP supports dynamic data structures better than Ada. LISP, C, and C++ provide constructs for pointing to objects of any type. Ada makes no guarantees, nor provisions, for pointing to arbitrary data objects. Portability can be a problem for any application that contains a dynamically defined database, a parser, or any other construct that operates on objects of different types.
- Variant records have three orthogonal aspects: dynamic/static, unconstrained/constrained, and mutable/immutable. Ada allows all combinations except for dynamic/unconstrained/mutable objects. Unfortunately, we need that combination. Dynamic objects do not need to specify list size at compile time. Unconstrained objects do not need to specify type at compile time. Mutable objects support one free list rather than separate free lists based on the discriminant. Unlike our earlier LISP design, we were forced to use dynamic/unconstrained/immutable objects. After the objects are no longer needed, they go on separate free lists based on their discriminant. However, we could run out of memory even though some of our free lists have unused objects. `Unchecked_Deallocation` routines are available, but Ada does not require that deallocation routines provide reused memory to the calling function.
- Incremental compilations dramatically reduce the compile and link times. Consequently, programmers can readily incorporate and test software changes. Most LISP workstations support incremental compilation. At the present time, Ada environments do not offer incremental compilation except for the Rational Ada environment.
- The LISP features that are difficult to replicate in Ada are: data interpreted as programs, a sophisticated input processor (reader), non-local exits (catch/throws), arbitrary precision arithmetic, and passing parameters to exception handlers.

Of the difficulties cited above, two were especially significant. Ada's lack of an inheritance capability resulted in repetitious code. Also, our Ada environment did not have the incremental compilation feature that was used to develop the earlier LISP version. However, our experience on the use of Ada was generally positive. We implemented some complicated functions in Ada and found the Ada version to be more reliable than the LISP version.